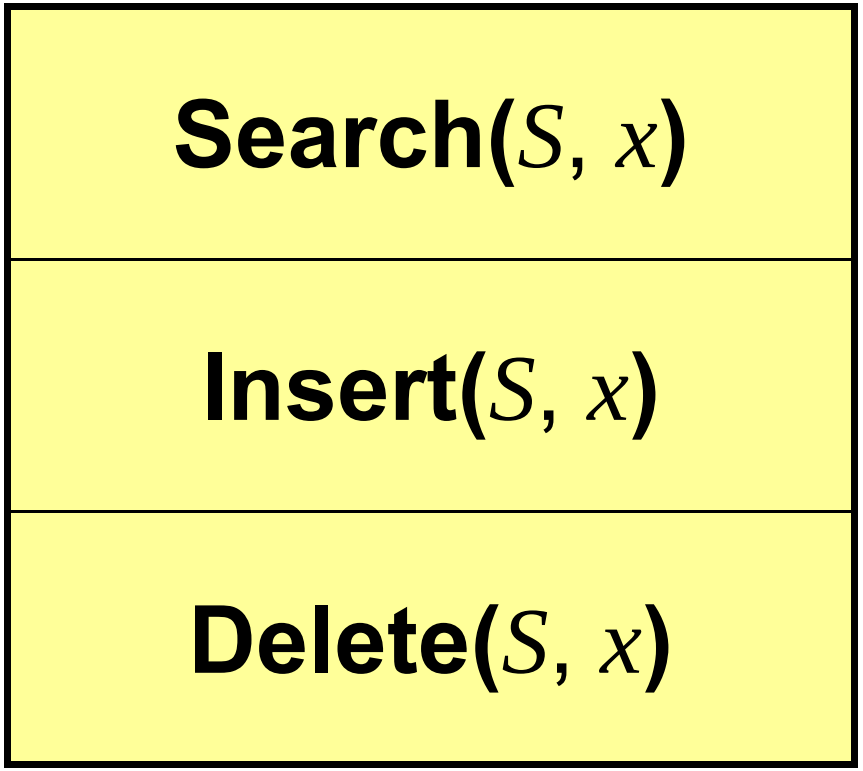
Algoritmer og Datastrukturer 1

Gerth Stølting Brodal

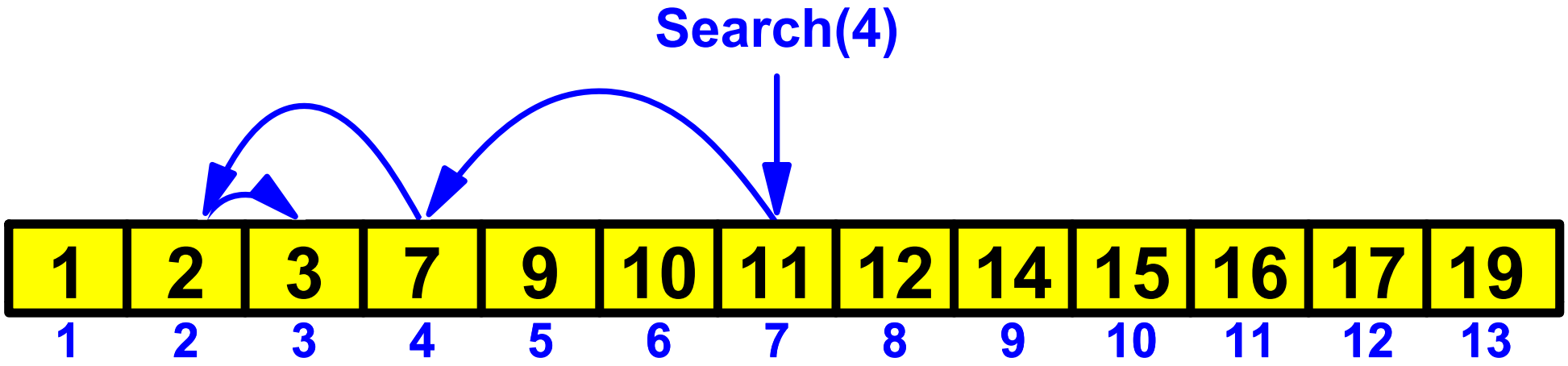
Binære Søgetræer [CLRS, kapitel 12]

AARHUS UNIVERSITET

Abstrakt Datastruktur: Ordbog

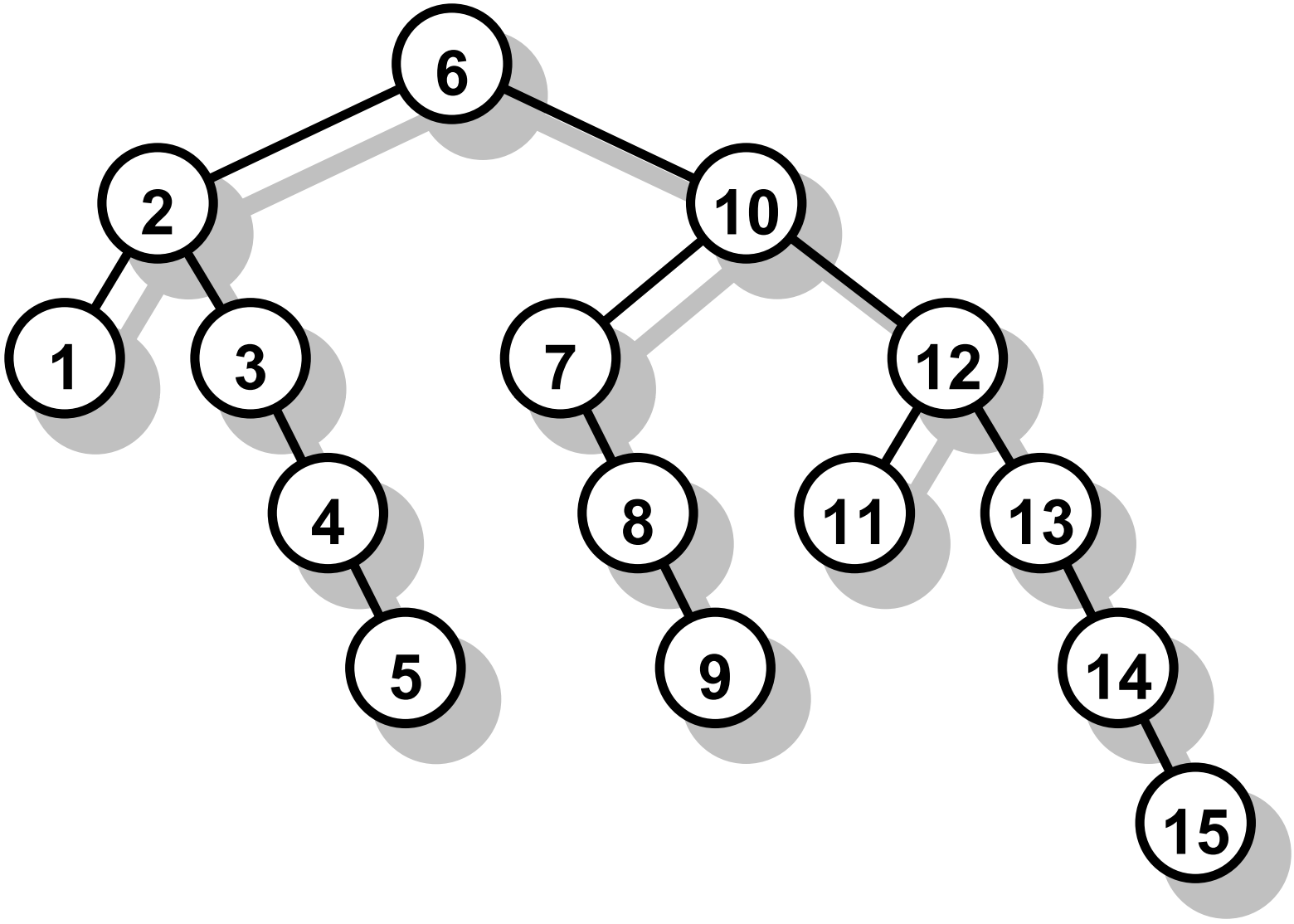


(Statisk) Ordbog : Sorteret Array



Search(S,x)	O(log n)
Insert(S,x)	O(n)
Delete(S,x)	

Søgetræ



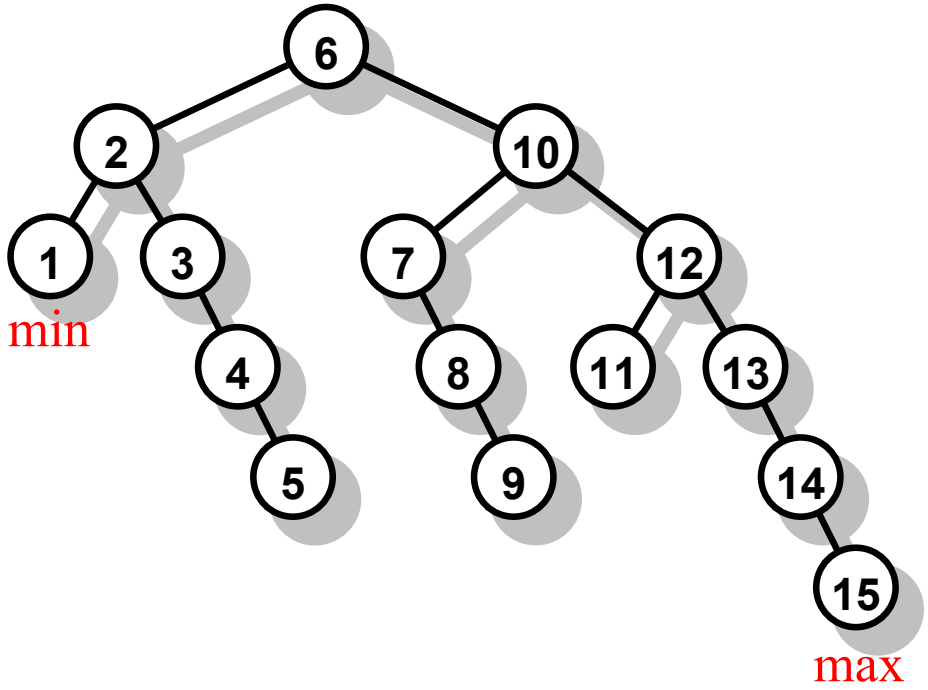
Invariant For alle knuder er elementerne i venstre (højre) undertræ mindre (større) end eller lig med elementet i knuden

Søgetræs søgninger

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2     then INORDER-TREE-WALK(left[x])
3          print key[x]
4          INORDER-TREE-WALK(right[x])
```

```
TREE-SEARCH(x, k)
1  if x = NIL or k = key[x]
2     then return x
3  if k < key[x]
4     then return TREE-SEARCH(left[x], k)
5     else  return TREE-SEARCH(right[x], k)
```

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ key[x]
2        do if k < key[x]
3              then x ← left[x]
4              else  x ← right[x]
5  return x
```



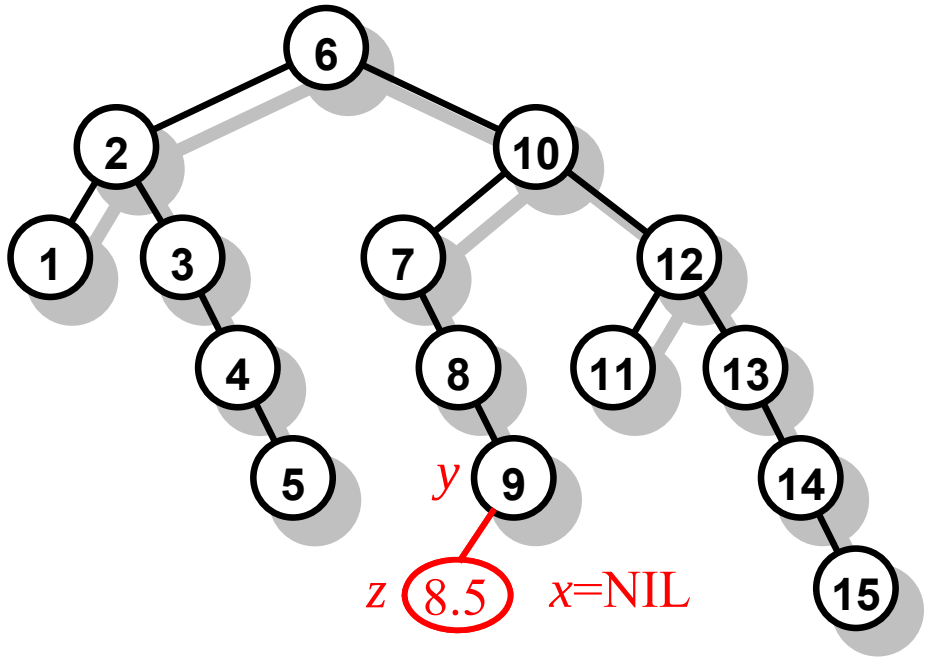

```
TREE-MINIMUM(x)
1  while left[x] ≠ NIL
2        do x ← left[x]
3  return x
```

```
TREE-MAXIMUM(x)
1  while right[x] ≠ NIL
2        do x ← right[x]
3  return x
```

Indsættelse i Søgetræ

```
TREE-INSERT(T, z)
 1   y ← NIL
 2   x ← root[T]
 3   while x ≠ NIL
 4         do y ← x
 5            if key[z] < key[x]
 6               then x ← left[x]
 7               else x ← right[x]
 8   p[z] ← y
 9   if y = NIL
10      then root[T] ← z
11      else if key[z] < key[y]
12              then left[y] ← z
13              else right[y] ← z
```

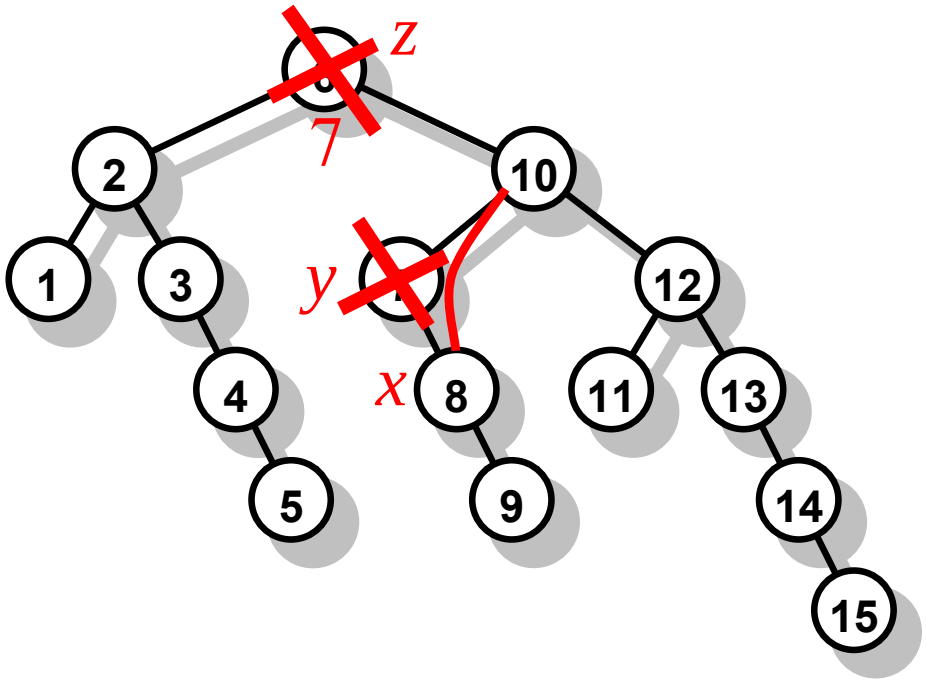
Iterativ søgning



▷ Tree T was empty

Slettelse fra Søgetræ

```
TREE-DELETE(T, z)
 1  if left[z] = NIL or right[z] = NIL
 2     then y ← z
 3     else y ← TREE-SUCCESSOR(z)
 4  if left[y] ≠ NIL
 5     then x ← left[y]
 6     else x ← right[y]
 7  if x ≠ NIL
 8     then p[x] ← p[y]
 9  if p[y] = NIL
10     then root[T] ← x
11     else if y = left[p[y]]
12             then left[p[y]] ← x
13             else right[p[y]] ← x
14  if y ≠ z
15     then key[z] ← key[y]
16          copy y's satellite data into z
17  return y
```




```
TREE-SUCCESSOR(x)
1  if right[x] ≠ NIL
2     then return TREE-MINIMUM(right[x])
3  y ← p[x]
4  while y ≠ NIL and x = right[y]
5        do x ← y
6           y ← p[y]
7  return y
```

Søgetræer

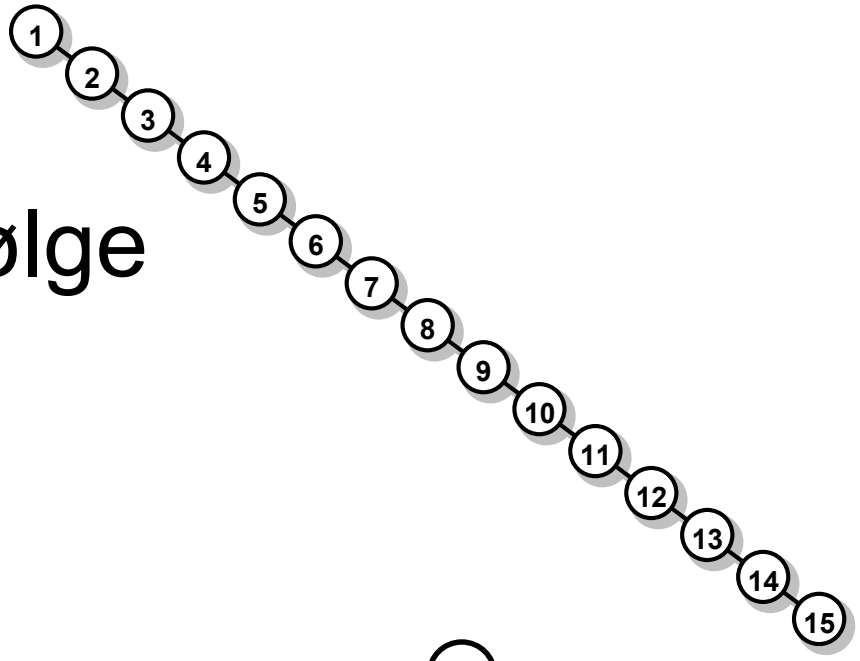
Inorder-Tree-Walk	$O(n)$
Tree-Search **Iterative-Tree-Search**	$O(h)$
Tree-Minimum **Tree-Maximum**	$O(h)$
Tree-Insert	$O(h)$
Tree-Delete	$O(h)$

h = højden af træet, n = antal elementer

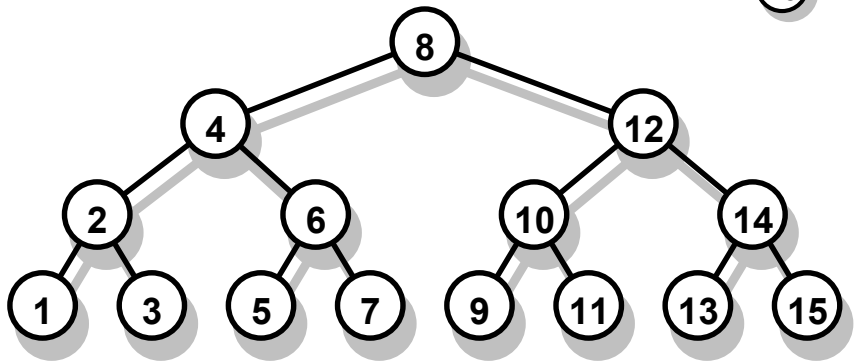
Højden af et Søgetræ ?

Største og Mindste Højde

Indsæt i stigende rækkefølge
- Højde n



Perfekt balanceret
(mindst mulig højde)
- Højde $1+\lfloor \log n \rfloor$



Tilfældige Indsættelser i et Søgetræ

Algoritme:

Indsæt n elementer i tilfældige rækkefølge

- Ligesom ved QuickSort argumenteres at den **forventede dybde** af et element er $O(\log n)$
- Den forventede **højde af træet** er $O(\log n)$, dvs. *alle knuder* har forventet dybde $O(\log n)$ [CLRS, Kap. 12.4]

Balancerede Søgetræer

Ved **balancerede søgetræer** omstruktureres træet løbende så søgetiderne forbliver $O(\log n)$

- AVL-træer
- BB[α]-træer
- Splay træer
- Lokalt balancerede træer
- Rød-sorte træer

- Randomized trees
- (2,3)-træer
- (2,4)-træer
- B-træer
- Vægtbalancerede B-træer
- ...

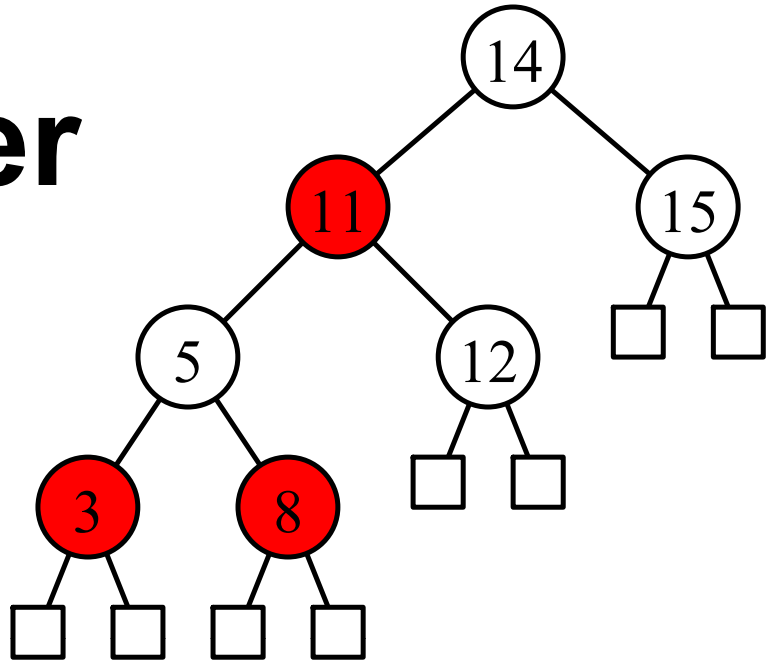
Algoritmer og Datastrukturer 1

Gerth Stølting Brodal

Rød-Sorte Søgetræer [CLRS, kapitel 13]

AARHUS UNIVERSITET

Rød-Sorte Søgetræer



Mål

Søgetræer med dybde $O(\log n)$

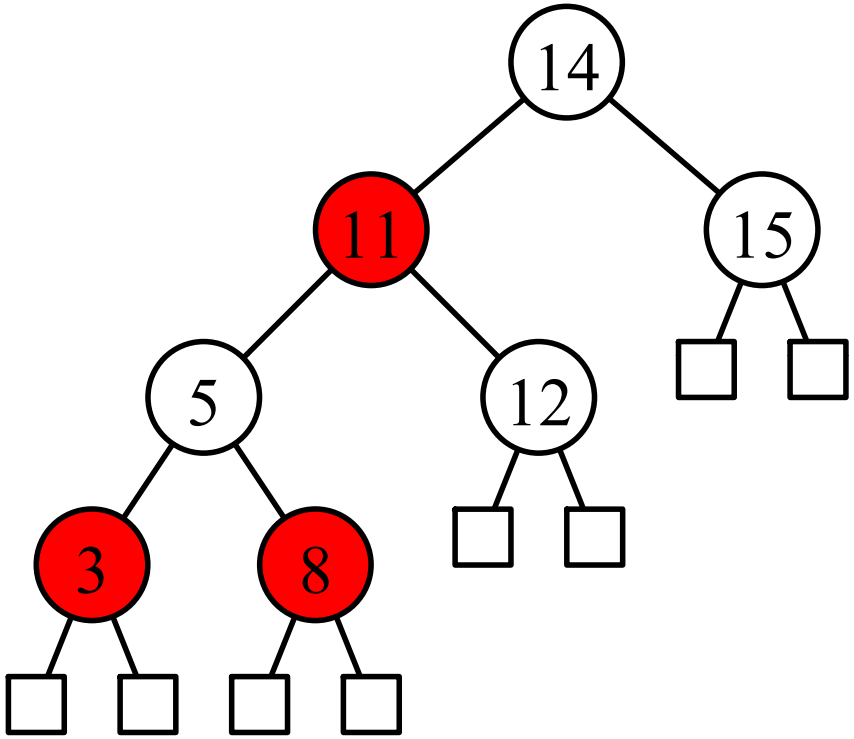
Invarianter

- Hver knude er enten **RØD** eller **SORT**
- Hver **RØD** knude har en **SORT** far
- Alle stier fra roden til et eksternt blad har **samme antal sorte knuder**

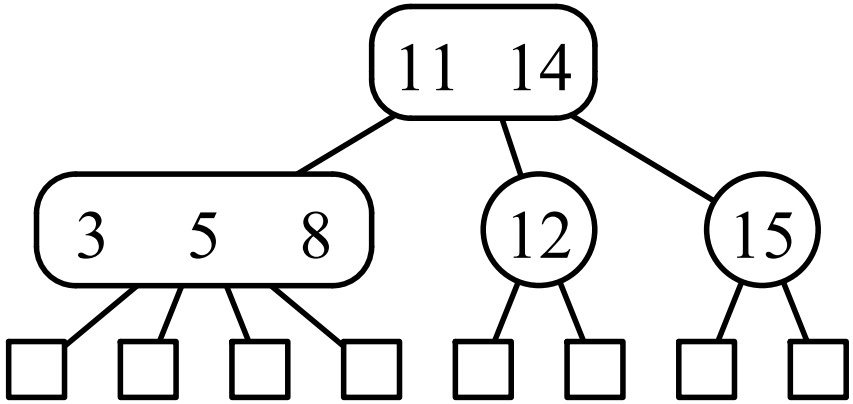
Sætning

Et rød-sort træ har højde $\leq 2 \cdot \log (n+1)$

Rød-Sorte vs (2-4)-træer



Rød-sort træ

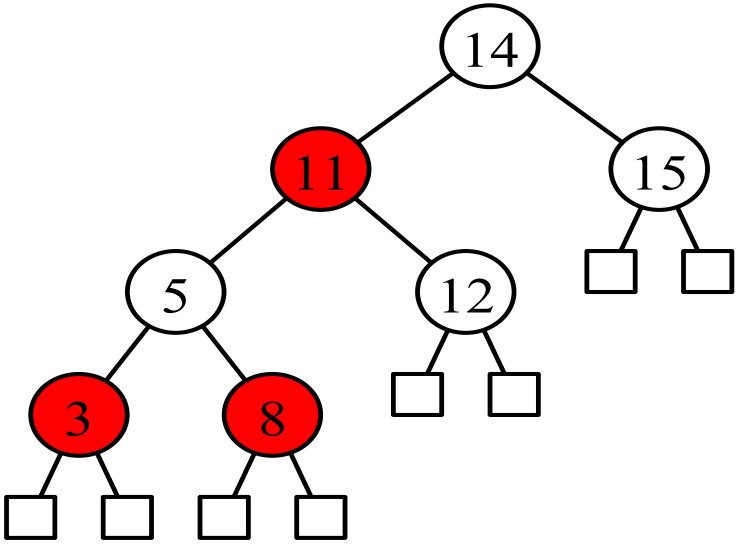


(2,4)-træ

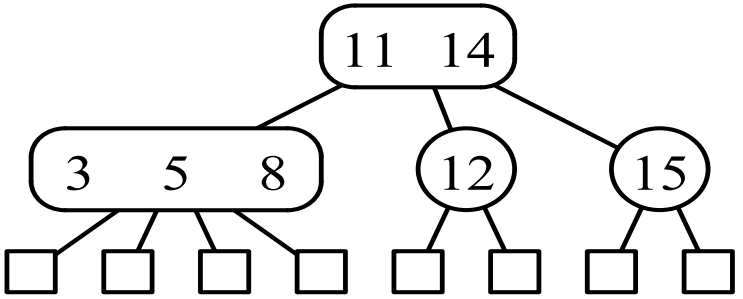
(2,4)-træ ≡ **rød-sort træ** hvor alle røde knuder er slået sammen med faderen

Egenskaber ved (2,4)-træer

- Alle interne knuder har mellem **2 og 4 børn**
- Knude med i **børn** gemmer i-1 **elementer**
- Stier fra roden til et eksternt blad er lige lange

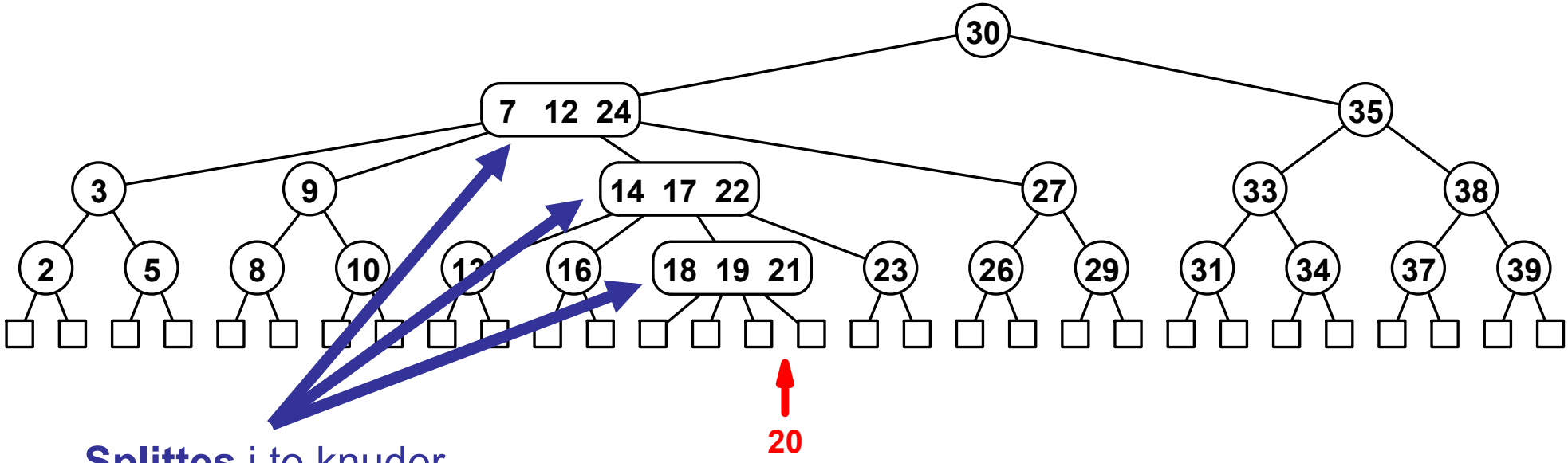


Rød-sort træ

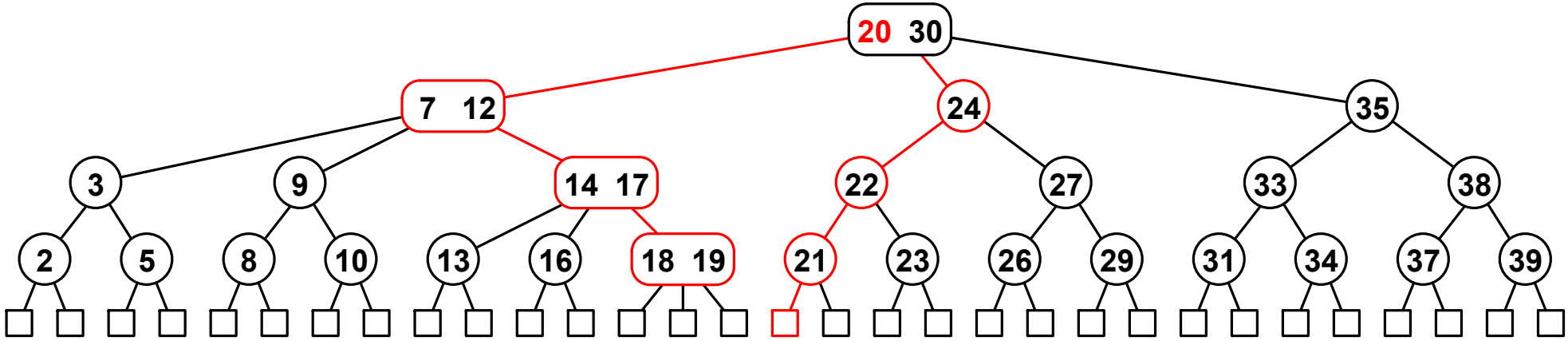


(2,4)-træ

Indsættelse i et (2,4)-træ

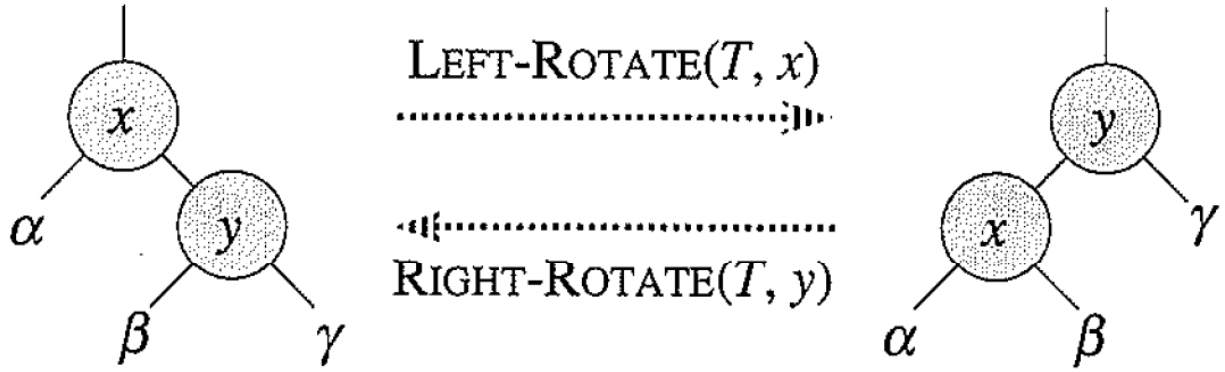


Splittes i to knuder
og midterste nøgle
flyttes et niveu op



Opdateringer af Rød-Sorte Træer

Rotationer



LEFT-ROTATE(T, x)

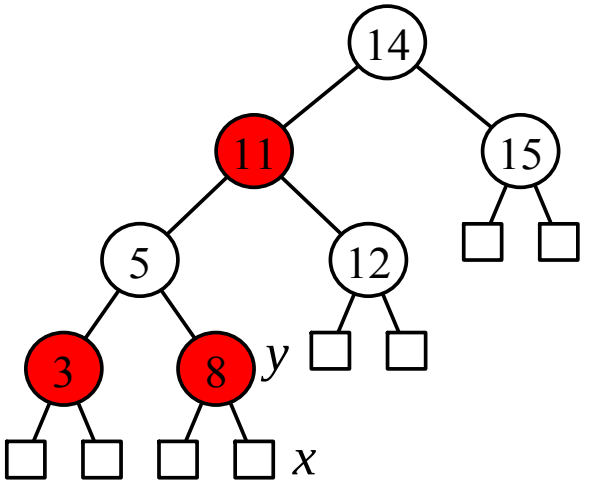
```
 1  y ← right[x]            ▷ Set y.
 2  right[x] ← left[y]      ▷ Turn y's left subtree into x's right subtree.
 3  if left[y] ≠ nil[T]
 4     then p[left[y]] ← x
 5  p[y] ← p[x]             ▷ Link x's parent to y.
 6  if p[x] = nil[T]
 7     then root[T] ← y
 8     else if x = left[p[x]]
 9             then left[p[x]] ← y
10             else right[p[x]] ← y
11  left[y] ← x             ▷ Put x on y's left.
12  p[x] ← y
```

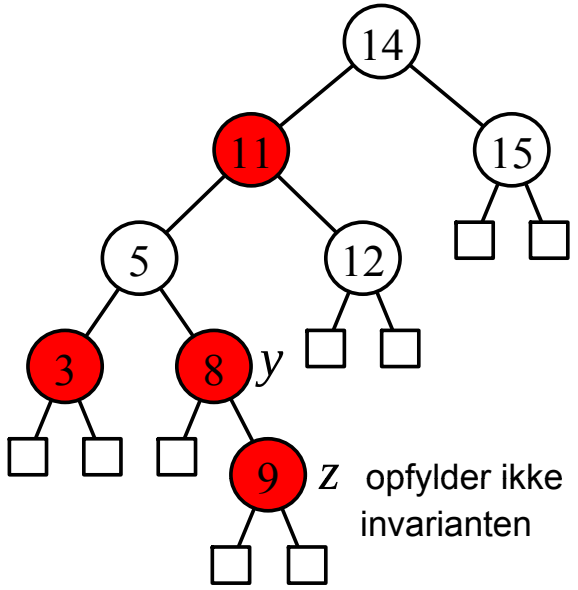
Insert

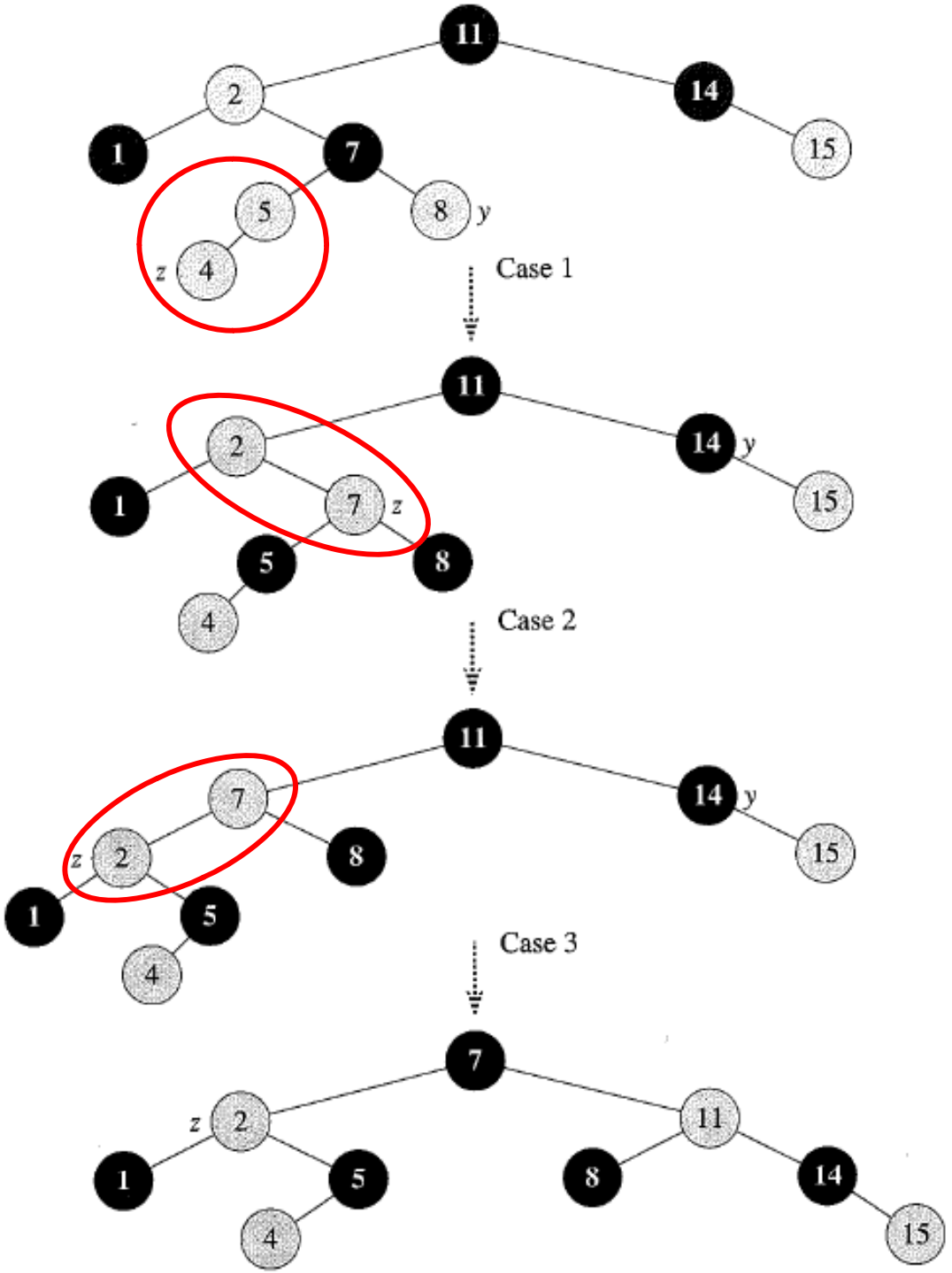
```
RB-INSERT(T, z)
 1  y ← nil[T]
 2  x ← root[T]
 3  while x ≠ nil[T]
 4        do y ← x
 5           if key[z] < key[x]
 6              then x ← left[x]
 7              else  x ← right[x]
 8  p[z] ← y
 9  if y = nil[T]
10     then root[T] ← z
11     else  if key[z] < key[y]
12              then left[y] ← z
13              else  right[y] ← z
14  left[z] ← nil[T]
15  right[z] ← nil[T]
16  color[z] ← RED
17  RB-INSERT-FIXUP(T, z)
```

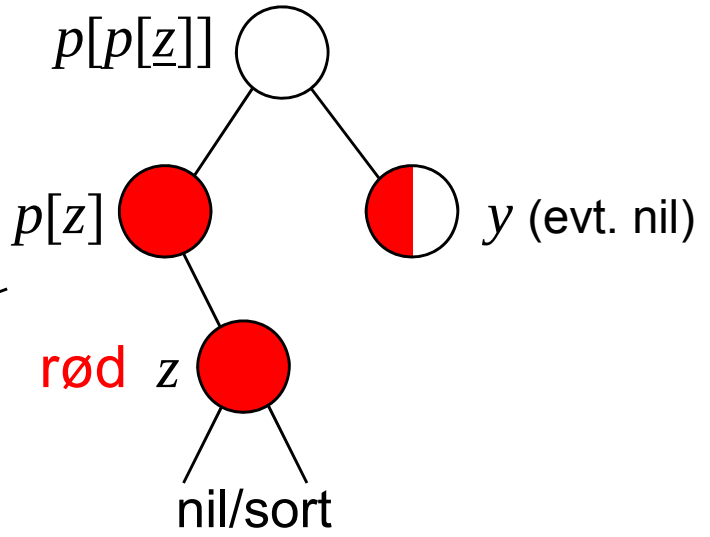
Ubalanceret indsættelse

Insert(9)





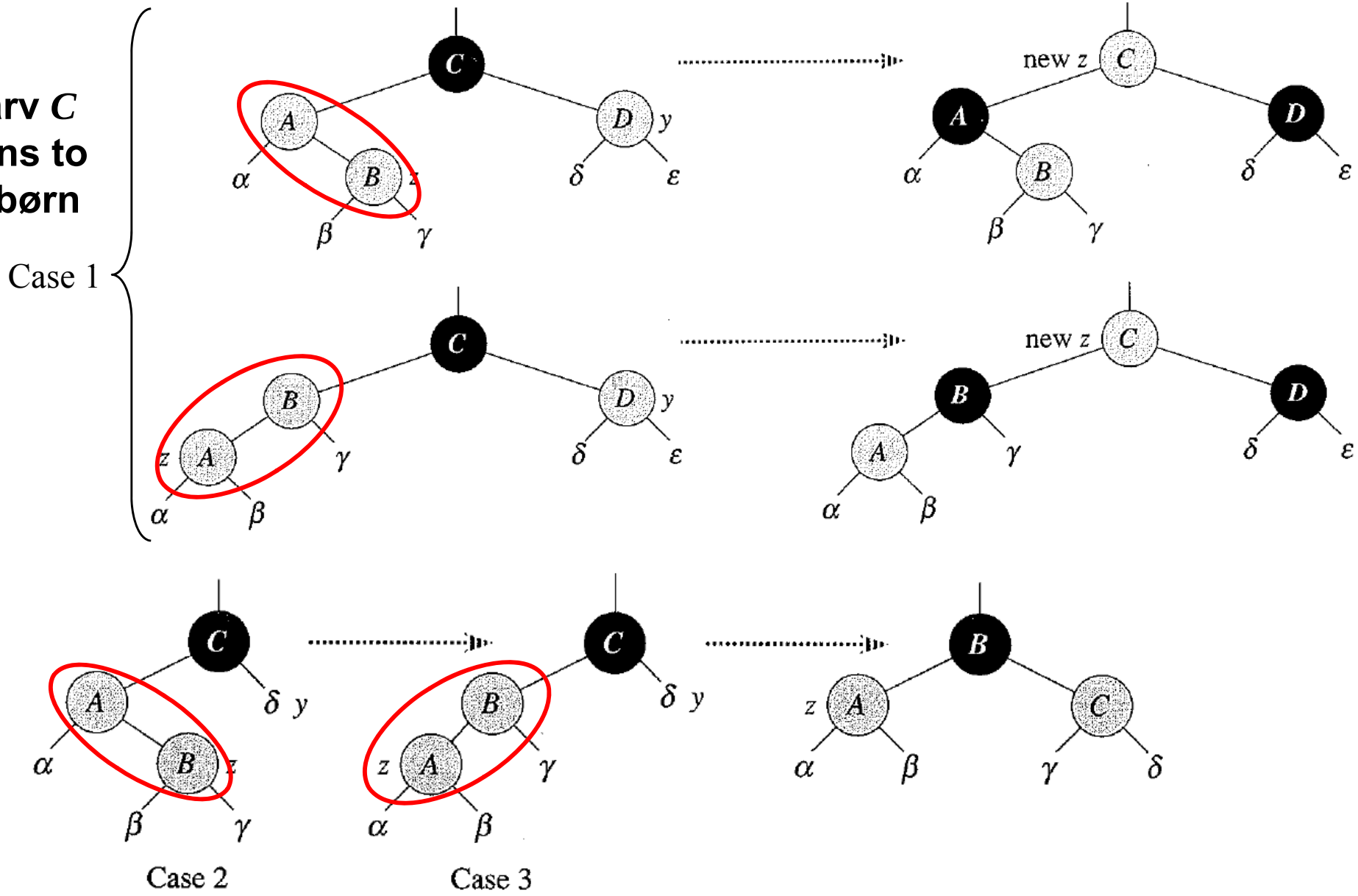

11
2
14
1
7
15
5
8
y
z
4
Case 1
11
2
14
y
1
7
z
15
5
8
4
Case 2
11
7
14
y
z
2
8
15
1
5
4
Case 3
7
z
2
11
1
5
8
14
4
15




```
RB-INSERT-FIXUP(T, z)
 1  while color[p[z]] = RED
 2        do if p[z] = left[p[p[z]]]
 3              then y ← right[p[p[z]]]
 4                   if color[y] = RED
 5                     then color[p[z]] ← BLACK                    ▷ Case 1
 6                          color[y] ← BLACK                       ▷ Case 1
 7                          color[p[p[z]]] ← RED                   ▷ Case 1
 8                          z ← p[p[z]]                            ▷ Case 1
 9                     else if z = right[p[z]]
10                            then z ← p[z]                       ▷ Case 2
11                                 LEFT-ROTATE(T, z)              ▷ Case 2
12                          color[p[z]] ← BLACK                   ▷ Case 3
13                          color[p[p[z]]] ← RED                  ▷ Case 3
14                          RIGHT-ROTATE(T, p[p[z]])              ▷ Case 3
15              else (same as then clause
                      with "right" and "left" exchanged)
16  color[root[T]] ← BLACK
```

Indsættelse i rød-sorte træer: rebalancering

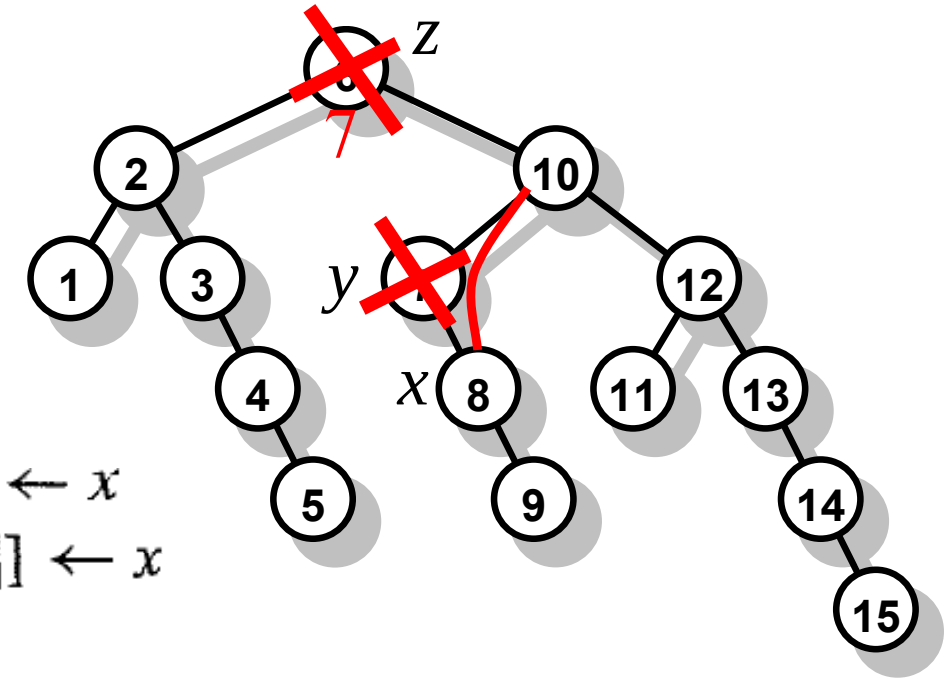
Omfarv C og dens to røde børn



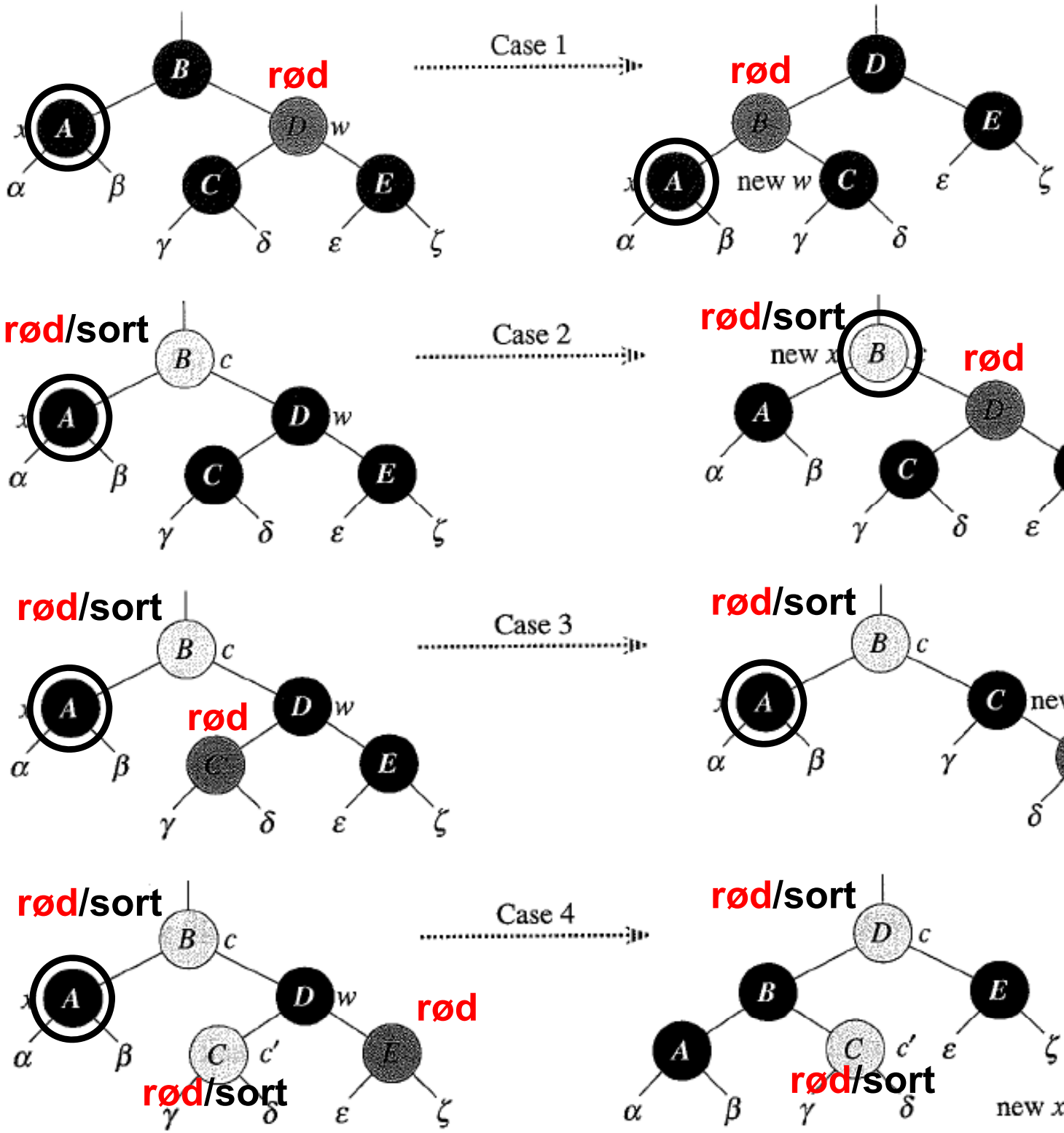
Delete

```
RB-DELETE(T, z)
 1  if left[z] = nil[T] or right[z] = nil[T]
 2     then y ← z
 3     else y ← TREE-SUCCESSOR(z)
 4  if left[y] ≠ nil[T]
 5     then x ← left[y]
 6     else x ← right[y]
 7  p[x] ← p[y]
 8  if p[y] = nil[T]
 9     then root[T] ← x
10     else if y = left[p[y]]
11             then left[p[y]] ← x
12             else right[p[y]] ← x
13  if y ≠ z
14     then key[z] ← key[y]
15          copy y's satellite data into z
16  if color[y] = BLACK
17     then RB-DELETE-FIXUP(T, x)
18  return y
```

Ubalanceret slettelse



```
RB-DELETE-FIXUP(T, x)
 1  while x ≠ root[T] and color[x] = BLACK
 2      do if x = left[p[x]]
 3          then w ← right[p[x]]
 4               if color[w] = RED
 5                  then color[w] ← BLACK                          ▷ Case 1
 6                       color[p[x]] ← RED                         ▷ Case 1
 7                       LEFT-ROTATE(T, p[x])                      ▷ Case 1
 8                       w ← right[p[x]]                           ▷ Case 1
 9               if color[left[w]] = BLACK and color[right[w]] = BLACK
10                  then color[w] ← RED                            ▷ Case 2
11                       x ← p[x]                                  ▷ Case 2
12                  else if color[right[w]] = BLACK
13                          then color[left[w]] ← BLACK            ▷ Case 3
14                               color[w] ← RED                    ▷ Case 3
15                               RIGHT-ROTATE(T, w)                ▷ Case 3
16                               w ← right[p[x]]                   ▷ Case 3
17                       color[w] ← color[p[x]]                    ▷ Case 4
18                       color[p[x]] ← BLACK                       ▷ Case 4
19                       color[right[w]] ← BLACK                   ▷ Case 4
20                       LEFT-ROTATE(T, p[x])                      ▷ Case 4
21                       x ← root[T]                               ▷ Case 4
22          else (same as then clause with "right" and "left" exchanged)
23  color[x] ← BLACK
```



Case 1 → 2,3,4

Case 2 → Problemet x flyttet tættere på roden

Case 3 → 4

Case 4 → Færdig

Dynamisk Ordbog : Rød-Sorte Træer

Search(S,x**)**	$O(\log n)$
Insert(S,x**)**	
Delete(S,x**)**	